Constraint Systems

Lab 01 - Python

Python Crash Course

General Information:

Python is a scripting language

- Object based
- Designed to be expressive and easy to read
- Compiled or interpreted?
  - There are multiple implementations
  - Default one: bytecode compilation + C interpreter

Probably the simplest "Hello world" ever:

```
print "Hello world!"
```

Python in Lab (for us)

- We will use Python via the excellent ipython console

```
ipython [--pylab] [script]
```

- Runs an advanced python console
- With the **`--pylab`** option: support for plotting and vector operations
- With optional argument: run a script

Give it a try!

- Open the console (no additional argument)
- Execute the line:

```
print 'My first python print'
```

Variables and Types

- Variables are loosely typed
- No variable-definition keyword

Some built-in types (those that matter for our course):

```
a = 10 # integer (typically C long)
b = 2.5 # real-valued numbers (typically C double)
c = 'A string with "quotes"'
d = "A string with 'apostrophes'"
e = True # A true boolean
f = False # A false boolean
g = None # Nothing (like "null" in Java)
```

Operators

Arithmetic operators:

```
a = 10 + 2 - 4 # sum and difference
b = 10 + (2 - 4) # change of priority (parantheses)
c = 10 * 2 # product
d = 10 / 3 # integer division
e = 10 % 3 # modulus (remainder of integer division)
f = 10 / 3.0 # real division
g = 10**2 # power
h = abs(-3.4) # absolute value
i = floor(3.4) # floor rounding
j = ceil(3.4) # ceil rounding
...
```

Operators

Logical operators:

```
a = v and w # Logical "and"
b = v or w # Logical "or"
c = not v # not
```

Comparison operators:

```
a = 5 < 10 # Less than
b = 5 <= 10 # Less than or equal
c = 5 == 10 # equal
e = 5 >= 10 # Greater than or equal
f = 5 > 10 # Greater than
g = 5 <= 7 <= 10 # Newbie error in C, works in python!
```

Operators

String formatting (C-style):

```
one = 1
two = 2.0
three = one + two
print 'And %d + %f = %.2f' % (one, two, three);
```

- `%d` prints an integer
- `%f` prints a real
- `%.2f` prints a real with two decimals
- `% (one, ...)` specifies the source for each field

Primitive Data Structures

Lists (mutable sequences):

```
a = ['a', 'is', 'a', 'list']
b = ['a', 1, True] # a list with multiple types
c = [1, 2, 3]

a[0] = 'b' # indexing (traditional)
print a[-1] # backwards indexing

a.append('another element') # append an element
a.pop() # pop last element
```

Primitive Data Structures

Lists have several other methods

Want to know which ones?

- Google "python list"
- Or use ipython on-line help!

First, built a list:

```
In [1]: a = [1, 2, 3]
```

Primitive Data Structures

Then, use tab completion:

```
In [2]: a.[HIT TAB]
a.append   a.count    a.extend   a.index...
```

Pick you favorite method and add a "?":

```
In [2]: a.count?[HIT ENTER]
```

And get some help!

```
Docstring: L.count(value) -> integer -- return ...
Type:      builtin_function_or_method
...
```

Primitive Data Structures

Tuples (immutable sequences):

```
a = ('a', 'is', 'a', 'tuple')
b = ('a', 1, True) # a tuple with multiple types

a[0] = 'b' # ERROR!!! Tuples are immutable
a = ('b', a[1], a[2], a[3]) # This works
print a[-1] # backwards indexing

print len(a) # number of elements
```

Primitive Data Structures

Tuple "superpowers" :-)

```
# tuple assignment
a, b = 10, 20

# tuple unpacking
c = (1, 2, 3)
d, e, f = c
# works with every iterable sequence!
d, e, f = [1, 2, 3]

# tuple printing
print d, e, f # automatic separator: space
```

There are a few other methods.

Primitive Data Structures

Dictionaries (i.e. maps):

```
a = {'name':'gigi', 'age':23} # key:value pairs
b = {'name':'gigi', 23:'age'} # keys of different types
# a key can be of any immutable type
c = {(0, 1):'a tuple as key!'}

print a['name'] # prints 'gigi'

print len(a) # number of items
print a.keys() # list of keys
print a.values() # list of values
print a.items() # list of (key, value) tuples
```

Many other methods!

Primitive Data Structures

Sets (with unique elements):

```
a = [1, 1, 2, 2, 3, 3]
b = set(a) # returns {1, 2, 3}
c = {1, 1, 2, 2, 3, 3} # builds {1, 2, 3}

b.add(4) # add an element
b.remove(3) # remove an element

print len(b) # number of elements
```

Other methods, as usual...

Primitive Data Structures

Special operators for collections:

```
a = [1, 2, 3] + [4, 5] # list concatenation
b = (1, 2, 3) + (4, 5) # tuple contatenation

print sum(a) + sum(b) # sum (lists and tuples)

print 2 in a # membership testing (any collection)

c = {'a':1, 'b':2}
print 'a' in c # "in" looks for keys in dictionaries
print 1 in c # so this is False
```

Control flow

- Instructions end when the line ends (no final ";")
- What if we need instructions on multiple lines?

```
print 1 + 2 + 3 + 4 + 5 + 6 + 7 + 9 + 10 + 11 + 12 + \
      13 + 14 # Use "\" to break a line
print (1 + 2 + 3 + 4 + 5 + 6 + 7 + 9 + 10 + 11 + 12 +
       13 + 14) # no need to do that in parentheses
```

- No "{}" block delimiters!
- Blocks are defined via indentation
- Practical examples in the next slides...

Control Flow

Conditional branches:

```
b = 10
if -1 <= a <= 1: # no parentheses needed
    b *= a;
    print 'b = %f' % b # no need for parantheses
                        # with a single term
elif a > 1:  # "elif"/"else" are optional
  print 'Big number' # diff. block, diff. indentation
else:
    print 'Small number'
```

- Bodies made of a single intruction can stay on the `if` line

```
if x > 0: print x
```

Loops

"For" loops:

```
for x in ['first', 'second', 'third']:
    print x

for i in range(10): # range returns [1,..., 9]
    if i == 0: continue # continue, as in C
    if i > 5: break # break, as in C
    print i

# enumerate returns a list of (index, item) tuples
# they can be "separated" via tuple unpacking
for i, x in enumerate(['three', 'item', 'list']):
    print i, x
```

Loops

"For" loops:

```
# "zip" returns a list of tuples
# range(x, y) = list of integers between x and y-1
for x, y in zip(range(5), range(5, 10)):
    print x, y # "0 5" "1 6" "2 7" ...
```

While loops:

```
a = 10
while a > 0:
    print a
    a -= 1 # decrement (no ++ or -- operators)
```

Comprehensions

A syntactical construct to build data structures on the fly

```
# list comprehension
a = [i**2 for i in range(5) if i % 2 == 0]

# set comprehension
b = {i**2 for i in range(5) if i % 2 == 0}

# dictionary comprehension (<key>:<expr>)
c = {i : i**2 for i in range(5) if i % 2 == 0}

# multiple levels
a = {(i,j) : 0 for i in range(10)
               for j in range(10)}
```

Comprehensions

General pattern (lists used as example):

```
<list comprehension> ::= [<generator expression>]
<generator expression> ::= <expr>
                           {for <var> in <collection>
                            [if <condition>]}
```

Functions

Function definition:

```
# Simple function
def f1(x):
    return x**2;

# Function within function
def f3(x):
    a = 10
    def f4(x): # scope includes that of f3
        return x*a; # f4 can access local vars of f3
    return f4(x)
```

Functions

Functions with named arguments (and default values):

```
def the_answer_is(res = 42):
    return res

print the_answer_is() # prints 42
print the_answer_is(13) # prints 13
print the_answer_is(res = 3) # prints 3
```

Functions

Functions are objects!

```
def transform(x, f):
    return [f(v) for v in x]

def g(x):
    return 2*x;

transform([1, 2, 3], g)
```

- Test it!
- You can use the magic **`%paste`** command in ipython

Functions

Functions as objects, an important case:

```
voti = [('gigi', 20), ('gianni', 30), ('gino', 24)]

def f(t): return -t[1]

# sorted returns a sorted copy of the collection
# key = score function to be used for sorting
for name, score in sorted(voti, key=f):
    print name # prints gianni, gino, gigi
```

Functions

Same as before, but more compact version:

```
voti = [('gigi', 20), ('gianni', 30), ('gino', 24)]

# sorted returns a sorted copy of the collection
# key = score function to be used for sorting
for name, score in sorted(voti, key=lambda t: -t[1]):
    print name # prints gianni, gino, gigi
```

Where:

```
lambda t: -t[1]
```

is a nameless function.

Modules

- Python has an extensive collection of external modules
- In fact, that's part of what makes Python cool
- We will see some of them during the course

For now, we need to know only **two things:**

- Modules are imported with:

```
import <module name>
```

- Our constraint solver is implemented as a module

Constraint Systems

Lab 1 - Google or-tools

Google or-tools

A software suite for solving combinatorial problems

- Developed by people @Google
- Main developed: Laurent Perron (formerly @ILOG)

Web site: https://developers.google.com/optimization/?hl=en

Several tools:

- CP solver **<-- our focus**
- LP solver (GLOP)
- SAT solver (BOP)
- MILP interface (CLP, CBC...)
- Custom Operations Research algorithms

Google or-tools

- The or-tools suite is written in C++
- Because it needs to be super-fast

However:

- C++ is not a good language for modeling (not expressive enough)
- So, wrappers were written for several languages:
  - Java
  - C#
  - Python

In the lab, we have installed the or-tools CP solver

Installing Google or-tools at Home

You can install Google or-tools at home

- Follow the python installation instructions at:

https://developers.google.com/optimization/

- Install from a binary release (simpler, more compact) or from source
- Some tools are available only if you install from source

In brief:

- You will need to obtain python (see next slide)
- Then download a file
- And then start a network-based installation process

Installing Google or-tools at Home

About obtaining Python:

On Linux (e.g. Ubuntu):

- Python is pre-installed
- Use the system package manager to get modules (e.g. ipython)

On OS X:

- I recommend installing via homebrew (https://brew.sh)

On Windows:

- I recommend downloading a distribution such as Anaconda:

https://www.continuum.io/downloads

Google or-tools Documentation

Google or-tools Documentation is quite scarce

- This is obviously bad
- There is a rough manual at:

https://developers.google.com/optimization/

- A reference manual at:

https://developers.google.com/optimization/reference/

Luckily, the API reference is good enough

- But there's a problem: it's just for C++!
- We will learn to "translate" the C++ API in python

Google or-tools Documentation

For us, the key parts of the API can be found at:

reference > constraint_solver > constraint_solver > Solver

And:

reference > constraint_solver > constraint_solver > IntVar

Let's see some golden rules for C++/Python translation...

or-tools Documentation: from C to Python

Solver API:

- Most C++ methods are available in python with the same name
- The methods that start with "Make" are available in python without the "Make" part. E.g. `MakeIntVar` becomes `IntVar`
- C++ vectors correspond to lists
- C++ string objects corresponds to string
- Most of the other translations are quite natural

IntVar API:

- Most of the methods are available in python with the same name

It's Easier with an Example

Download & unzip the start-kit associated to this lecture

- File **`lab01-ortools-ref.py`** contains a very simple CSP modeled and solved with or-tools
- Try to solve the problem with:

```
python lab01-ortools-ref-py
```

Or with:

```
ipthon
run lab01-ortools-ref-py
```

The script should print a list of solutions

Let's see the structure of an or-tools model...

Structure of an or-tools Model

First, we have to import the or-tools module:

```
from ortools.constraint_solver import pywrapcp
```

Then we have to build an instance of a solver object:

```
slv = pywrapcp.Solver(model_name)
```

Then we can build variables:

```
x1 = slv.IntVar(min, max, name) # API 1
x2 = slv.IntVar(values, name) # API 2
```

Structure of an or-tools Model

For building constraints, the python API offers a trick:

- Many operators (e.g. `+, -, <, ==, ...`) have been redefined
- If one term of the operator is a variable, then:
  - "`+, -, *, ...`" build an expression
  - "`<, <=, ==, >=, >, !=`" build a constraint

So we can build constraints like:

```
x + 2 <= y
x != y
x = 2*y - 3
```

Structure of an or-tools Model

Once a constraint has been built, it must be added to the solver:

```
svl.Add(constraint)
```

- If we forget about this, the constraint is not propagated

Then, this part of the script defines how to do search:

```
decision_builder = slv.Phase(all_vars,
                             slv.INT_VAR_DEFAULT,
                             slv.INT_VALUE_DEFAULT)
```

- We will not modify this part (for now)
- One important thing: the first argument is a list, with the variables to be considered by DFS

Structure of an or-tools Model

Finally, we have the code to:

- Initialize the search process:

```
slv.NewSearch(decision_builder)
```

- Trigger search until a solution is found:

```
while slv.NextSolution():
  # here we can print the solution data, e.g.
  print 'The value of x is %d' % x.Value()
```

- Tear down the search data structures:

```
slv.EndSearch()
```

Constraint Systems

Lab 1 - (Finally) Time to Practice!

Two Problems

- Try to get some familiarity with python and the or-tools API...
- ...by modeling and solving two problems:

Problem 1: Map Coloring

- Our good old map coloring problem, for the whole of Italy
- You can find a template script in `lab01-ex1.py`
- Try to solve the problem for different number of colors

What is the minimum number needed to color each region in Italy?

Two Problems

Problem 2: A Puzzle by Lewis Carrol

- There are five houses.
- The Englishman lives in the red house.
- The Spaniard owns the dog.
- Coffee is drunk in the green house.
- The Ukrainian drinks tea.
- The green house is immediately to the right of the ivory house.
- The Old Gold smoker owns snails.
- Kools are smoked in the yellow house.
- Milk is drunk in the middle house.
- The Norwegian lives in the first house.
- …

Two Problems

Problem 2: A Puzzle by Lewis Carrol

- The man who smokes Chesterfields lives in the house next to the man with the fox.
- Kools are smoked in the house next to the house where the horse is kept.
- The Lucky Strike smoker drinks orange juice.
- The Japanese smokes Parliaments.
- The Norwegian lives next to the blue house.

Who owns a zebra and who drinks water?

- A template script can be found in `lab01-ex2.py`